Urychlení běhu vlákna

Motivace

- Aby to běželo rychleji
- Aby vícevláknový výpočet běžel co nejrychleji, interakce mezi jednotlivými procesory musí být co nejmenší
  - Část interakce vědomě zavádí programátor použitím synchronizačních primitiv jako je např. kritická sekce
  - Část interakce vzniká na úrovni hardware
    - Pokud programátor má povědomí o tom, co se děje na úrovni hardware, má šanci omezit i tuto část interakce

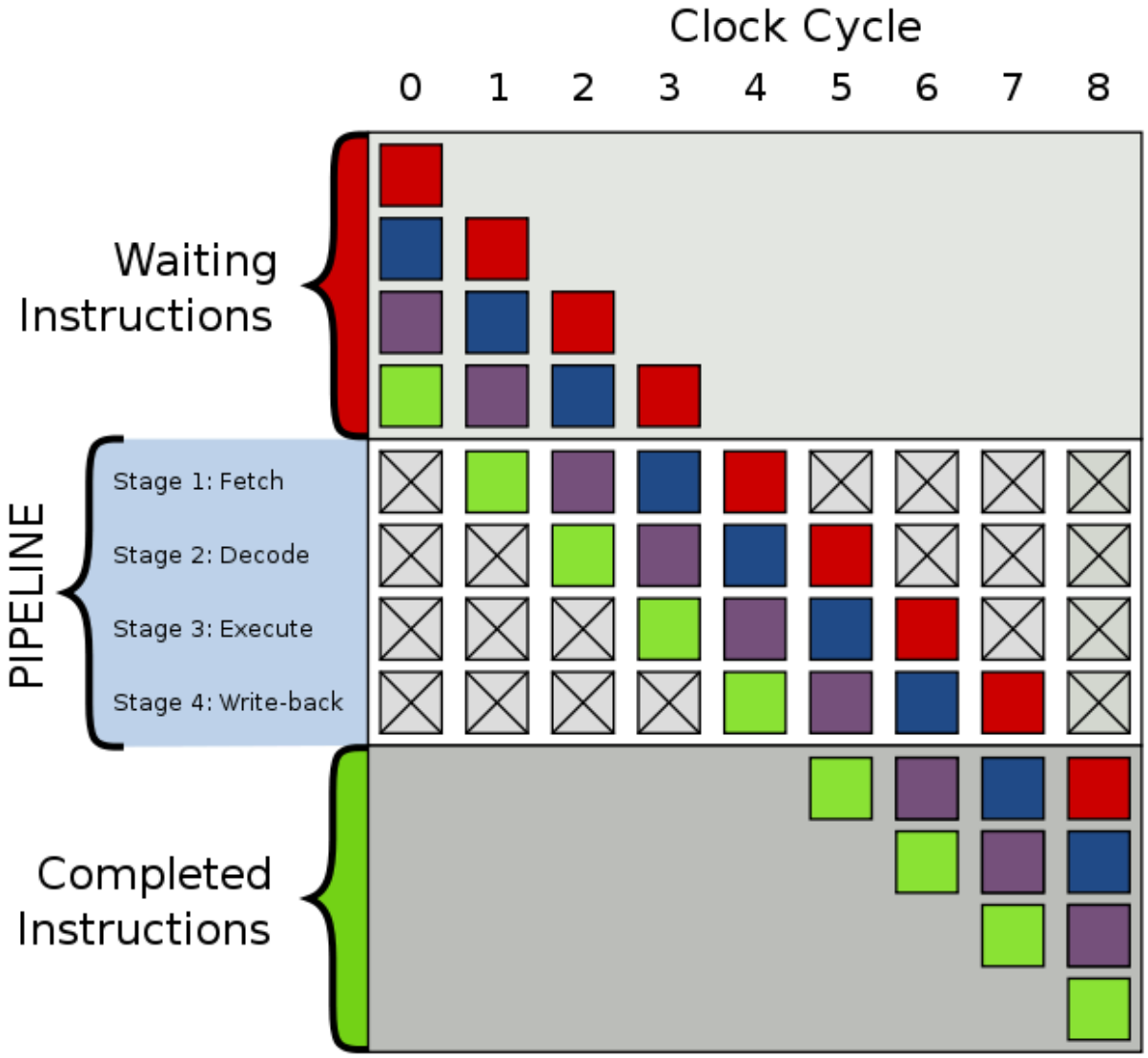
- Koupit si výkonnější hardware umí každý. Napsat efektivní, a přesto stále čitelný kód umí jen málokdo.

Pipeline

- Architektura procesoru mj. říká, jaké má procesor instrukce
  - Tj. to, s čím pracuje programátor

- Zpracování jedné instrukce lze rozdělit do několika fází jako je načtení, dekódování, vykonání, přístup do paměti
  - Počet fází závisí na konkrétním procesoru
  - Instrukce se skládá z několika mikroinstrukcí
    - => mikroarchitektura, kterou programátor nevidí
- procesor bez pipeline vykonává jednu instrukci několik cyklů – např. 386 (byť ne že by neměla nic jako pipeline)

- pipeline procesoru (486) má sama o sobě možnost vykonávat instrukce paralelně – tj. nezávisle na sobě se snaží vykonávat různé mikroinstrukce
  - paralelismus na úrovni instrukcí
  - ve špičkovém výkonu lze vykonat jednu instrukci za jeden cyklus



http://en.wikipedia.org/wiki/File:Pipeline,_4_stage.svg

- moderní superskalární procesor (586+) používá další techniky, které mu umožní vykonat více než jednu instrukci za jeden cyklus
  - a pak se projeví znatelný rozdíl mezi kódem, který byl a který nebyl optimalizován
    - velkou část umí dobrý překladač, takže platí že, optimalizovat se vyplatí tak dlouho, dokud to není na úkor čitelnosti kódu
    - v kódu se pak programátor stará o to, aby překladač a procesor měli „volné ruce“ k provádění svých optimalizací

      - je-li cílem vysoký výpočetní výkon, zapomeňte na JIT překladače – stojí další čas a výpočetní výkon, aby se nakonec možná dobrali ke kódu, který lze získat už statickým překladem

      - finální optimalizace, které mají na svědomí celkový výkon, se totiž odehrávají v procesoru

Spekulativní multithreading

- při zvyšování počtu instrukcí za cyklus už může docházet k hazardům, takže se jejich vykonávání někdy uměle zpožďuje
  - stejně jako už dříve nešlo donekonečna zvyšovat pracovní frekvenci procesoru
- možným dalším směrem vývoje je analýza nezávislosti instrukcí procesorem, zda by nešly vykonávat ve vlastních vláknech (pokročilejší out-of-order execution)
  - tj. jakási hw autoparalelizace sériového kódu

Predikce skoků

```
int a=10; int fact=1;
do {fact=fact*a; a--;} while (a>=0);
```

```
  mov edx, 0ah   ; a = 10
  mov r8d, 1     ; fact = 1

@jump:           ; do
  imul r8d,edx   ; fact=fact*a
  sub edx, 1     ; a--
  jns @jump      ; while (a>=0), tj. podmíněný skok
```

- vykonává-li procesor instrukce paralelně, pak se při podmíněném větvení programu dostane do bodu, kdy:
  - buď počká na dokončení ostatních instrukcí, aby mohl vypočítat podmínku větvení
    - trvá to dlouho, ale nesplete se
  - anebo podmínku větvení odhadne
    - trvá to krátce, ale je tu penalizace, když se splete

- když procesor špatně odhadne podmínku větvení, pak musí zahodit všechny výpočty, které udělal na základě chybné predikce a vyprázdnit pipeline – tj. udělat rollback
  - je to drahé

- moderní procesor si udržuje historii (Branch Target Buffer - BTB), kolikrát na dané podmínce skočil, a podle toho dělá predikci, zda opět skočí
  - tím minimalizuje riziko penalizace

- Nemá-li procesor pro podmínku záznam v Branch Target Buffer, pak použije Static Branch Prediction Algorithm

  - `if (cond){<procesor předpokládá cond==true>};`
  - `do { <procesor předpokládá opakování smyčky> } while (cond);`

  - Pokud procesor nezná příslušnou podmínku, tak se jí naučí
    - U podmínky řídící proměnné cyklu se tedy může splést úvodem
    - Při mnoha špatně navržených podmínkách, může docházet ke zdržení, když budou podmínky špatně vyhodnocované
      - Špatně napsaný program

    - Programátor může podmínky navrhnout tak, aby vyhovovali Static Branch Prediction Algorithm
      - Z tohoto vychází domněnka, že JVM může profilací kódu adaptivně přeuspořádat podmínky za běhu programu tak, aby program běžel co nejrychleji
        - A programátor se s tím nemusel zatěžovat

      - Skutečnost je ovšem taková, že s podmínkami se lze vypořádat elegantnějším způsobem, který už JVM nemůže zlepšit
        - JVM tedy dokáže překonat C++ překladač tehdy, jedná-li se o špatně napsaný program!

Podmíněné větvení

- Podmínka if je sekvence instrukcí cmp a jne/je
- Uvažujme kód, který něco dělá s jednotlivými parametry

```
for i:=0 to n-1 do
  if cond1 then DoThing1(params[n])
     else if cond2 then DoThing2(params[n])
       else DoThing3(params[n]);
```

- Vykonávání procedur DoThingX může modifikovat proměnné cond1 a cond2
  - Zatím ale předpokládejme, že to neplatí, a pak je lepší

```
if cond1 then
  for i:=0 to n-1 do DoThing1(params[n])
    else if cond2 then
      for i:=0 to n-1 do DoThing2(params[n])
      else for i:=0 to n-1 do DoThing3(params[n]);
```

- Ale ještě lepší by to bylo s využitím pointerů

```
var doThing:TDoThing;

begin
  if cond1 then doThing:=@DoThing1
     else if cond2 then doThing:=@DoThing2
        else doThing:=@DoThing3;

  for i:=0 to n-1 do
    doThing(@param[n]);
```

- Protože i kdyby DoThingX měnily podmínky condX, lze kód lehce modifikovat tak, aby změnil obsah proměnné doThing
- Výsledek tak redukuje Branch Target Buffer trashing
  - Obsazení bufferu podmínkami, které lze eliminovat

- Jak vypadá původní kód v x86-64
  - Ve zjednodušené verzi bez optimalizace

```
    sizeofparam EQU 48              ;sizeof(Param)
                                    ;aligned to 8
    xor rsi, rsi                    ;n

@Repeat:
    cmp BYTE PTR [cond1],0          ;if cond1 then
    je @Test2                       ;else

    lea rbx, [params]               ;ptr to 1st param
    lea rax, [rbx+rsi*sizeofparam]
    PushParam
    call DoThing1
    add rsp, sizeofparam            ;caller cleans-up
    jmp @Finish

@Test2:
    cmp byte ptr [cond2], 0
    je @Else

    lea rbx, [params]               ;ptr to 1st param
    lea rax, [rbx+rsi*sizeofparam]
    PushParam
    call DoThing2
    add rsp, sizeofparam            ;caller cleans-up
    jmp @Finish

@Else:
    lea rbx, [params]               ;ptr to 1st param
    lea rax, [rbx+rsi*sizeofparam]
    PushParam
    call DoThing3
    add rsp, sizeofparam

@Finish:
    inc rsi
    cmp rsi, [n]
    jne @repeat
```

```
PushParam macro
    mov rcx, sizeofparam
    shr rcx, 3

@dopush:
    push qword ptr [rax]
    add rax, 8
    dec rcx
    jnz dopush
endm
```

- Jak po optimalizaci cyklu vypadá nový kód v x86-64
  - Opět ve zjednodušené verzi
  - Vlastní kód cyklu je mnohem kratší

```
    sizeofparam EQU 48              ;sizeof(Param)
    ptrsize     EQU 8               ;sizeof(pointer)
    DoThing     DB 0 DUP(ptrsize);null pointer

    cmp BYTE PTR [cond1], 0         ;or QWORD PTR if
    je @DoThing2                    ;evaluted frequently

    lea rax, [DoThing1]
    mov [DoThing], rax
    jmp @Cycle

@DoThing2:
    cmp BYTE PTR [cond2], 0
    je @DoThing3

    lea rax, [DoThing2]
    mov [DoThing], rax
    jmp @Cycle

@DoThing3:
    lea rax, [DoThing3]
    mov [DoThing], rax

@Cycle:
    mov rcx, [n]                    ;n
    or  rcx, rcx                    ;faster test for 0
    jz  @Finish                     ;on unsigned n
    lea rax, [params]               ;ptr to 1st param

@Repeat:
                                    ;rax is ptr to param
                                    ;fastcall => no push
    call QWORD PTR [DoThing]        ;calling the proc
    add rax, sizeofparam            ;next param
    dec rcx                         ;dec rcx a jnz @ lze
    jnz @Repeat                     ;nahradit loop, ale
@Finish:                            ;loop má větší režii
```

Vyhodnocování podmínek

- Překladač obvykle vyhodnocuje komplexní podmínku ve zkrácené formě, je-li to možné
- Seřadit podmínky tak, aby podle pořadí vyhodnocování
  - Byly co nejmenší nároky na získání hodnot potřebných k vyhodnocení podmínky
    - proměnné v registru vs. proměnné v paměti
    - proměnné vs. volání funkcí
    - O() funkcí vracejících porovnávané hodnoty
  - Došlo co nejrychleji k vyřazení co největšího počtu možností
    - např. v databázi pacientů by podmínka pohlaví s ~50% úspěšností vyřadila nehledaný subjekt
- Opakování matka moudrosti
  - Sekvence AND končí s prvním false
  - Sekvence OR končí s prvním true

Ternární operátor a CMOV

- Rodina instrukcí x86
  - Konkrétně u cmov jde o přiřazení, pokud je splněna podmínka
- Zejména v x86-64 kódu se používá u překladu ternárního operátoru – viz eliminace skoků
- Je třeba si uvědomit, kdy je rychlejší skok a kdy ternární operátor

- Pokud je podmínka snadno predikovatelná, např. řídící proměnná cyklu, pak se vyplatí dělat if
  - Procesor se snadno naučí správně odhadnout, kdy bude splněna a plně využije svého výkonu

- Pokud je ale podmínka závislá na vstupních datech, pak
  - Pokud jsou data predikovatelná, např. nějak uspořádaná, pak je lepší použít if
    - Např. budeme-li hledat v poli prvků hodnot nejmenší prvek
    - Např. budeme-li v uspořádaném poli bytů určovat, zda je byte větší než 0x80 nebo menší
  - Pokud jsou ale vstupní data taková, že daná podmínka bude ne/splněna s takovým rozložení pravděpodobnosti, které se blíží rovnoměrnému, pak je efektivnější použít ternární operátor – cmov
    - Např. budeme-li v **ne**uspořádaném poli bytů určovat, zda je byte větší než 0x80 nebo menší

- Datová závislost ternárního operátoru/cmov
  - x <- cmov (x, y) - vzniká datová závislost, tj. zpoždění a je lepší použít if
  - z <- cmov (x, y) – nevzniká datová závislost a co použít závisí na pravděpodobnosti splnění podmínky

- U if lze pravděpodobnost specifikovat
  - GCC: __builtin_expect
  - MSVC: __assume

- Skoky lze sice eliminovat, ale pozor na to, že
  - Buď může dojít k nárůstu kódu za skok, který lze snadno predikovat, což se projeví zpomalením
  - Anebo také může vzniknout datová závislost, která se opět projeví nežádoucím zpomalením
  - => tj. skoky se neliminují za každou cenu

Eliminace skoků podle Intelu

- Zdroj: Intel® 64 and IA-32 Architectures Optimization Reference Manual
- Eliminace skoků má zásadní vliv na zvýšení výkonu

Se skoky	Bez skoků

`cmp a, b`
`jbe L30`
 	
`mov ebx const1`
`jmp L31`
 	
`L30:`
`mov ebx, const2`
 	
`L31:`
	`xor ebx, ebx ;výsledek`
	`cmp a, b`
	`setge bl ;if ge then bl:=1`
	
	`;get all bits set to 1 or 0 in ebx`
	`sub ebx, 1 ;ebx=11...11 or 00...00`
	
	`and ebx, C3; C3=const1 – const2`
	`;ebx je teď buď nula, nebo rozdíl`
	
	`;takže už jen stačí přičíst const2`
	`add ebx, CONST2`
	`;ebx=const1 nebo const2`

`test ecx, ecx`
`jne 1H`
`mov eax, ebx`
 	
`1H:`
	`test ecx, ecx ;Test the flags`
	`cmoveq eax, ebx ;If the equal flag is`
	`;set, move`
	`;ebx to eax- the 1H:`
	`;tag no longer needed`

Se skoky	S jedním skokem (until)

`for i:=1 to 100 do`
`begin`
`if i mod 2=0 then`
`a[i]:=0`
`else a[i]:=1;`
`end;`
	`i:=100;`
	`repeat`
	`a[i]:=0;`
	`a[i-1]:=1;`
	`dec(i, 2);`
	`until i=0;`

- Využití cmov – bytecode pro to nemá ekvivalent
  - Tj. ztrácí se důležitá informace od programátora
  - A právě díky agresivnímu využívání cmov instrukcí Intelovský překladač obvykle vítězí v testech

Eliminace skoků podle AMD

- The AMD Athlon™ 64 and AMD Opteron™ processors have the capability to cache branchprediction history for a maximum of three near branches (CALL, JMP, conditional branches, or returns) per 16-byte fetch window.

- Data-dependent branches acting upon basically random data cause the branch-prediction logic to mispredict the branch about 50% of the time.

Naivně (se skokem)	Lépe (bez skoku)

`int abs(int x) {`
`  if (x<0) x-=x;`
`  return x;`
`}`
	`mov ecx, [x]  ; Load value.`
	`mov ebx, ecx  ; Save value.`
	`neg ecx  ; Negate value.`
	`cmovs ecx, ebx ; If negated value`
	`  ; is negative,`
	`  ; select value.`
	`mov [x], ecx  ; Save labs result.`

`uint min(uint x,`
`  uint y){`
`  return x < y ?`
`  x : y;`
`}`
	`mov eax, [x]  ; Load x value.`
	`mov ebx, [y]  ; Load y value.`
	`cmp eax, ebx  ; EBX <= EAX ?`
	`  ; CF = 0 : CF = 1`
	`cmovnc eax, ebx ; EAX = (EBX <=`
	`  ; EAX) ? EBX : EAX`
	`mov [z], eax  ; Save min(X,Y).`

- Při rekurzivním, podmíněném volání podprogramu, od hloubky 12 může dojít ke špatnému odhadu adresy, tj. vyhodnocení, kam se skočí při návratu z podprogramu.

Naivně (rekurze)	Lépe (iterace)

`long fac(long a) {`
`if (a == 0) { return 1; }`
`else {`
`myp(a); //Can cause`
`//returns to be`
`//mispredicted`
`return (a * fac(a - 1));`
`}`
`}`
	`long fac(long a) {`
	`long t = 1;`
	`while (a > 0) {`
	`myp(a);`
	`t *= a;`
	`a--;`
	`}`
	`return (t);`
	`}`

- Při vytváření switch/case záleží, jestli jsou možnosti sousední hodnoty, nebo jestli jsou mezi nimi mezery
  - Pokud jsou mezery, překladače switch převedou na sérii porovnávání
  - Snažit se eliminovat mezery

Mezery	Sousední

`switch (grade) {`
`case 'A': ... break;`
`case 'B': ... break;`
`case 'C': ... break;`
`case 'D': ... break;`
`case 'F': ... break;`
`}`
	`switch (grade) {`
	`case 'A': ... break;`
	`case 'B': ... break;`
	`case 'C': ... break;`
	`case 'D': ... break;`
	`default: ... break;`
	`}`

- Lze totiž sestavit tabulku ofsetů, kam se má skočit v konkrétních case
- A z podmínky lze potom vypočítat index to této tabulky
- Pokud to nejde, switch bude jedna velká série podmíněných skoků

Mapování (a inkrementování pointerů)

```
procedure byte2hex(const val:byte; outbuf:pchar);
const conv:array[0..15] of char = (0123456789ABCDEF);
begin
  outbuf^:=conv[val shr 4];
  inc(outbuf);
  outbuf^:=conv[val and $0F];
  inc(outbuf);
  outbuf^:=#0;
end;
```

- outbuf je už alokovaná paměť, kam zapsat výsledek
- Připomenutí: procesor obsahuje branch-prediction,
  - Spekulativně se vykonává kód dopředu podle toho, jaký se předpokládá výsledek skoku
  - Uvedená konverze nemění stav branch-prediction
    - Tj. neovlivňuje urychlování volající funkce
    - Kdyby se alokovala paměť, např. pro nový string, tak už to stav branch-prediction ovlivní

- Pole může být naplněno i za chodu a obsahovat např. adresy rutin; např. pro:

```
if (param = 1) or (param = 7) then proc1
  else if param in [4..5] then proc2;
```

- Je efektivnější udělat case/switch
  - Záleží na hodnotách, a jak překladač case zrealizuje

```
case param of
  1, 7: proc1;
  4..5: proc2;
end;
```

- Anebo (zejména když by byl více než jeden parametr)

```
var procs:array[1..7] of TProc;
begin
  FillChar(procs, sizeof(procs), #0);
  procs[1]:=@proc1;
  procs[4]:=@proc2;
  procs[5]:=@proc2;
  procs[7]:=@proc1;

  ...
  //a namísto case
  procs[param];
end;
```

- Co bude rychlejší, to už záleží na konkrétním kódu
  - A co vlastně budou větve case/switch obsahovat

Náročnost operací

- Díky optimalizacím mikroarchitektury se zdá, že provedení instrukce trvá kratší dobu, než tomu ve skutečnosti je
- Ve skutečnosti stále každá instrukce trvá 1 až několik cyklů, jako u 386
- Proto je třeba vyhýbat se drahým/dlouhotrvajícím instrukcím
- Má-li procesor vykonat drahou instrukci, musí zpozdit ostatní instrukce v pipeline
- Proto se např. opakované dělení provádí násobením inverzním číslem
  - `ix:=1.0/x; y:=a*ix; z:=b*ix;`
  - U integerů, `m:=i div (j*k)` namísto `m:=(i div j)div k`
    - Dělení hard-coded číslem se už ale nechává na překladači, který pro to má vlastní triky

Vektorizace a Loop Unrolling

- Pro procesor s vektorovými instrukcemi lze loop unrolling chápat jako nápovědu pro překladač s auto-vektorizací
  - V nejhorším dojde k většímu využití pipelines procesoru

Naivně	Lépe

`double a[100], sum;`
`int i;`
 	
`sum = 0.0f;`
 	
`for (i = 0;`
`  i < 100; i++) {`
`  sum += a[i];`
`}`
`//Překladač to může,`
`//ale i také nemusí`
`//správně pochopit.`
 	
`//Loop-unrolling`
`//také snižuje počet`
`//porovnávání, tj. i`
`//případných skoků.`
	`double a[100], sum;`
	`double sum1, sum2, sum3, sum4;`
	`int i;`
	
	`sum1 = 0.0f;`
	`sum2 = 0.0f;`
	`sum3 = 0.0f;`
	`sum4 = 0.0f;`
	
	`for (i = 0; i < 100; i + 4) {`
	`  sum1 += a[i];`
	`  sum2 += a[i+1];`
	`  sum3 += a[i+2];`
	`  sum4 += a[i+3];`
	`}`
	`sum = (sum4 + sum3) +`
	`  (sum1 + sum2);`

- Použití i++ v a[…] by mohlo vnést závislost, tj. zhoršit výkon při využití pipelines

Počet iterací při Loop Unrolling

- Vektorové operace mají operandy o fixní velikosti
- Překladač tak potřebuje vědět, kolik potenciálně vektorizovatelných operací jde po sobě
  - Aby věděl, jestli může použít vektorové instrukce

- Následující kód jde vektorizovat

```
for (i = 0; i < 100; i++) {
  src[i] += dst[i];
}
```

- Následující kód jde také částečně vektorizovat
  - Lze ho rozdělit na dvě části, podle velikosti n
  - První bude vektorizovaná, druhá už ne

```
for (i = 0; i < (n & ~3); i++) { src[i] += dst[i]; }
for (i = (n & ~3); i < n; i++) { src[i] += dst[i]; }
```

- Následující kód už ale ne, protože nevíme, kolikrát smyčka proběhne

```
while (*dst) {
  *src += *dst;
  src++;
  dst++;
}
```

- Následující kód už lze opět vektorizovat
  - Ale je napsán se znalostí dat, kterou překladač nemá

```
while (*dst) {

  *src += *dst;
  *(src+1) += *(dst+1);
  *(src+2) += *(dst+2);
  *(src+3) += *(dst+3);

  src += 4;
  dst += 4;
}
```

- Např. s GCC lze ověřit, jak dopadla autovektorizace pomocí přepínače -ftree-vectorizer-verbose

Out-of-Order Execution & Registry Renaming

- Uvažujme následující kód

```
R1 = mem[addr1]
R1 = R1 + 4
mem[addr1] = R1

R1 = mem[addr2]
R1 = R1 + 8
mem[addr2] = R1
```

- Oba bloky nelze provést paralelně, protože používají stejný registr
- Pokud bychom použili další registr, pak už je můžeme provést paralelně

```
R1 = mem[addr1]          R2 = mem[addr2]
R1 = R1 + 4              R2 = R2 + 8
mem[addr1] = R1          [addr2] = R2
```

- Ve specifikaci architektury registrového procesoru jako je x86 existuje konečný počet pojmenovaných registrů
- Překladač se sám snaží využít registry tak, aby se co nejvíce instrukcí dalo provést paralelně

- Procesor se snaží o další dvě optimalizace
  - Out-of-order execution – procesor sice načítá instrukce v pořadí daném překladačem, ale prochází je, a určuje jim nové pořadí, ve kterém je vykoná
    - Snaží se detekovat jejich vzájemné závislosti tak, aby je mohl čekání na dokončení jedné instrukce překrýt vykonáváním jiné, nezávislé instrukce

  - Registry renaming – procesor má více registrů, než kolik jich je pojmenovaných na úrovni architektury
    - V případě, kdy překladači dojdou volné registry, procesor je stále schopný paralelně vykonávat instrukce, které zdánlivě pracují se stejným registrem – viz první případ

    - Dalším pozitivem registry renaming je i to, že se může vyplatit loop-unrolling i tehdy, když se nejedná o kód převeditelný do vektorových instrukcí

    - Nicméně bude stále platit, že čím méně proměnných ve funkci, tím více proměnných dokáže překladač uchovat pouze v registrech

      - Což zrychlí běh aplikace a vyplatí se „roztrhat“ např. náročnější tělo cyklu na několik cyklů

Paměťová závislost

- U instrukcí jako je x <- cmov (x, y) vzniká datová závislost
- Existuje však ještě paměťová závislost, kdy instrukce load (from memory) musí čekat na instrukci store (to memory)
  - Přičemž přístup k paměti se může stát úzkým hrdlem

- Řešením je eliminace nepotřebných sekvencí Store-and-Load používání dočasných proměnných, které mohou být realizovány pomocí registru
  - Programátor by měl pomoci překladači jejím použitím – namísto co nejkratšího zápisu

  - Naivně

```
double x[VECLEN], y[VECLEN];
unsigned int k;

for (k = 1; k < VECLEN; k++) {
  x[k] = x[k-1] + y[k];
}
```

  - Lépe

```
double x[VECLEN], y[VECLEN];
unsigned int k;
double t;

t = x[0];
for (k = 1; k < VECLEN; k++) {
  t = t + y[k];
  x[k] = t;
}
```

Falešná závislost

- Registrů na úrovni mikroarchitektury je konečný počet
- A nakonec se musí výsledek zapsat někam, kde je viditelný z pohledu architektury
- Příkladem, na x86 existují čtyři způsoby, jak zapsat do registru nulu
    - registr eax se běžně používá pro předávání návratových hodnot a parametrů funkcí

Instrukce	**Strojový kód**
mov eax, 0	b8 00 00 00 00
and eax, 0	83 e0 00
sub eax, eax	29 c0
xor eax, eax	31 c0

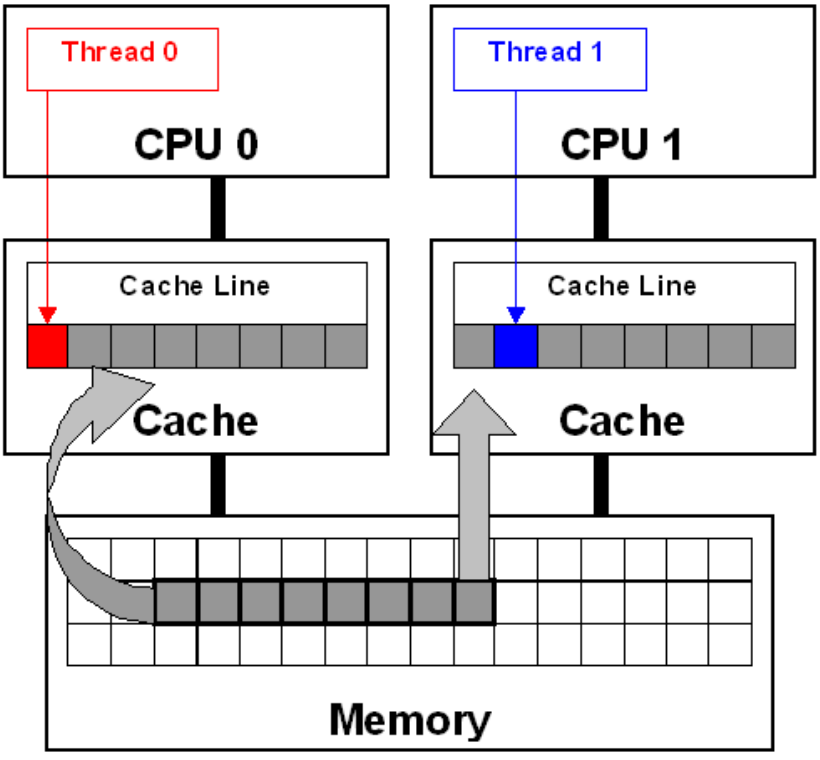
- Krátký kód je dobrý, protože se ho pak více vejde do cache
    - Tj. vyřadíme mov a and

- sub a xor představují datovou závislost
    - jenže u xor si je procesor vědom, že jde o falešnou závislost
        - a překladač to ví

- => nesnažte se přechytračit překladač, o cílovém procesoru toho víc daleko více než vy – jenom mu pomozte pro-něj čitelným zápisem

Falešné sdílení

- False Sharing
- Pozor, je to něco jiného než falešná závislost
- Každý procesor má svou vlastní cache, ale paměťový systém musí zajistit tzv. cache-coherence – konzistenci dat
- False sharing nastane tehdy, když různé procesory (a každém z nich běžící jiné vlákno) modifikují různé proměnné, které ale sdílejí stejnou cache-line
    - Ačkoliv se může jednat o proměnné, které nejsou sdílené mezi vlákny, přesto je daná cache-line zneplatněna
        - Následně je vynucena její aktualizace => a dojde ke zpomalení
        - Paradoxem je, že sériovému programu se to nestane a může tak být i rychlejší, než špatně napsaný paralelní program.



https://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS) {
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

    #pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

    #pragma omp atomic
    sum += sum_local[me];
}
```

- U sum_local může dojít k false sharing, protože pole může být tak malé, aby se celé vešlo do jedné cache-line
- Možným, ale špatným, řešením by bylo zvýšit velikost proměnné na tolik bytů, aby se do cache-line vešla vždy jen jedna proměnná
  - Jenže to vyžaduje znalost cílového procesoru
    - Velikost cache-line (tj. velikosti bloku dat) může být 32, 64, ale i 128
  - Rychle se tím neužitečně obsazuje prostor cache, který není až zase tak velký

- Lepším řešením je, aby si každý thread vytvořil svou pracovní kopii data, která pak zapíšou pouze jedno, až na závěr svého výpočtu

```
void threadFunc(void *param)  {
    ThreadParams *p = (ThreadParams*) param;
    auto local = p->variable;
    for(local=p->start;local<p->end;local++) {
        // Function computation
    }

    p->variable = local; // Update only once
}
```

Transakční paměť

- Mějme datovou strukturu naplněnou např. 1024 prvky
- A chceme ji modifikovat z více vláken
- Nejjednodušší je použít jeden zámek, protože vzniká čitelný kód a je nejmenší riziko chyby synchronizace
- Jenomže bude-li chtít např. první vlákno modifikovat desátý prvek a současně druhé vlákno 600. prvek, jedno z nich bude muset počkat, ačkoliv nehrozí riziko poškození dat

- Softwarovým řešením je použít více zámků (rozdělit prvky do několika regionů, každý s vlastním zámkem), takže se sníží pravděpodobnost, že na sebe budou dvě vlákna čekat, budou-li modifikovat nezávislé prvky
  - Riziko ovšem nelze eliminovat úplně a navíc se tím zvyšuje složitost synchronizačního kódu
    - => zvyšuje se riziko chyby

- Hardwarovým řešením je transakční paměť
  - Procesor ji nedělá automaticky, ale poskytuje speciální instrukce
    - Programátor pak buď využije knihovny svého jazyka, které pracují s transakční pamětí
      - Např. gnu libc, Intel TBB
    - Anebo v jazycích jako je C++ ji může použít sám

- Cílem transakční paměti je poskytnou výhody jednoho zámku při rychlosti sw řešení s několika zámky

- Programátor v kódu označí instrukci, kterou začíná kritická sekce
  - Touto instrukcí zároveň označí část kódu, kam procesor skočí v případě, že se transakci nepodařilo uskutečnit – tzv. fallback path

- Procesor zjistí začátek kritické sekce a od té chvíle jsou veškeré zápisy do paměti lokální, dokud:
  - Není transakce úspěšně dokončena a pak se provede commit
  - Anebo dojde ke konkurenčnímu zápisu do hlídané oblasti paměti
    - Pak procesor skočí na první instrukci fallback path
    - Velikost hlídané paměti je záležitostí aktuálního procesoru, např. 32kB L1 cache u Intel Haswell

- x86-64 umí dva režimy
  - HLE (hardware lock ellision) – kompatibilní se starým kódem, znovupoužití prefixů instrukcí
  - RTM (restricted transaction mode) – nové instrukce

- HLE – modifikace spinlocku

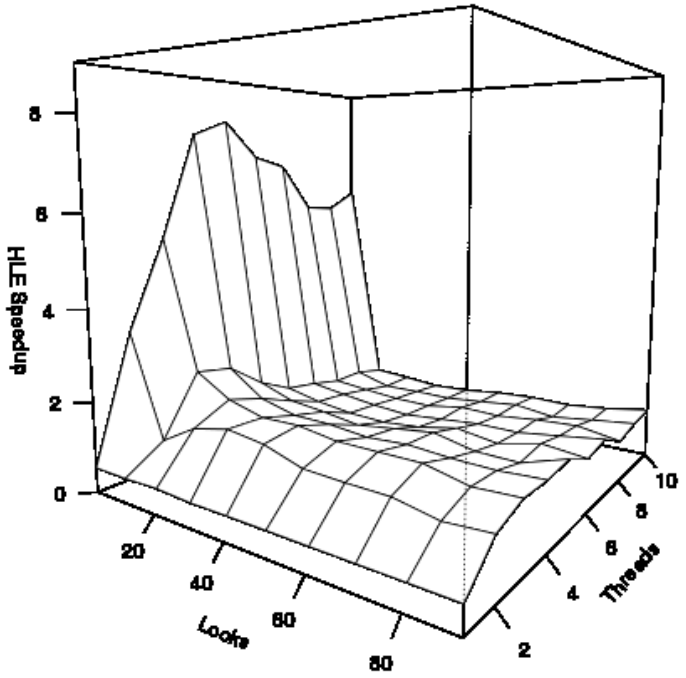
```
      mov rdx, qword (-1);hodnota zamčeno
      xor rax, rax ; hodnota odemčeno
spin: xacquire lock cmpxchg8b [zámek], rdx
      jz spin ;if (rax == [zámek]) ZF=1
      ;zápis do kritické sekce dle potřeby
      xrelease mov qword ptr[zámek], 0
```

  - fallback path začíná instrukcí cmpxchg – ta díky xacquire napoprvé proběhne, jako kdyby nebylo zamčeno

- RTM – modifikace spinlocku

```
        mov rdx, qword (-1);hodnota zamčeno
        xor rax, rax ; hodnota odemčeno

        xbegin spin
        jmp locked

spin:   lock cmpxchg8b [zámek], rdx
        jz spin ;if (rax == [zámek]) ZF=1

locked:
        ;zápis do kritické sekce dle potřeby

        mov qword ptr[lock], 0    ; odemknout
        xend
```

  - Instrukce xbegin říká, kde začíná fallback path
  - Na rozdíl od HLE jsme ušetřili lock cmpxchg8b
  - Existuje i instrukce xabort
  - Registr eax je při skoku do spin nastaven na hodnotu kódu, proč transakce neprošla



http://brooker.co.za/blog/2013/12/14/intel-hle.html

Paměť

Využití cache

- Viz superlineární urychlení

- Výkonnosti programu lze pomoci, pokud se data zpracovávají po takových částech, aby zůstala v RAM, nejlépe aby se kód, příp. data, dostala do cache procesoru

- Např. u zpracovávání objemných dat lze očekávat, že OS odstránkuje část dat z RAM na disk
  - Řešením je zpracovávat data po takových částech, kdy nám celá část zůstane v RAM

- Programátor neovlivní, kolik z jeho kódu bude v cache procesoru, ale může zvýšit pravděpodobnost tohoto efektu malým kódem, schopným samostatné činnosti
  - Co nejmenší kód není vždy nejvýhodnější, někdy je lepší větší kód, protože může běžet výrazně lépe

- U operací nad datovými strukturami je dobré implementovat jen to, co je nezbytné
  - Např. pokud pro seznam není nutné použít double-linked list, použije se single-linked list, a tím se zredukuje výsledný programový kód
  - Návyk používat hotové balíčky, než bych chvíli programoval, je dobrá cesta ke cache-trashingu

  - I datová struktura má svou režii, díky kterému pak zbývá o to méně místa v cache na užitečná data
    - Je pro malé množství položek nutný strom?
    - Od jakého počtu položek se už strom vyplatí?

Pointer aliasing

- Na jednu proměnnou může ukazovat více pointerů a překladači to potom brání v optimalizaci
- Příkladem mohou být pointery na pole
  - Je-li překladači známa velikost pole, můžou proběhnout 2 zápisy do paměti souběžně, nepřekrývají-li se
  - Ale není-li u typu pointeru uvedena maximální velikost pole, překladač musí předpokládat možnost pointer-aliasing
- Zdroj: SW Optimization Guide for AMD64 Processors

```
typedef struct { float m[4][4]; } MATRIX;

void XForm(float *res, const float *v,
           const float *m, int numverts) {
float dp;
int i;
const VERTEX* vv = (VERTEX *)v;

for (i = 0; i < numverts; i++) {
  dp  = vv->x * *m++;
  dp += vv->y * *m++;
  dp += vv->z * *m++;
  dp += vv->w * *m++;
  *res++ = dp; // Write transformed x.
```

- Kvůli adresováním paměti pomocí *m musí překladač předpokládat, že *m může ukazovat kamkoliv a nemá tak moc možností, jak optimalizovat
- Pokud se ale použije zápis přes indexy, tak už je v tom pro překladač nápověda, kam se ukazuje

- Adresování paměti lépe:

```
for (i = 0; i < numverts; i++) {
  rr->x = vv->x * mm->m[0][0] + vv->y * mm->m[0][1] +
          vv->z * mm->m[0][2] + vv->w * mm->m[0][3];
```

Alokace paměti

- Pokud možno, alokovat pracovní buffery jednou a dost
- Častá alokace na haldě == špatná alokace
- Operační systém používá nějaké bloky, po kterých přiděluje paměť
  - Říkat si o méně než je velikost bloku je neefektivní
- A co když chceme víc, než kolik se vejde do bloku?
  - Je třeba alespoň nějak znát chování programu a očekávanou velikost
  - Jde o to, abychom nežádali OS o alokaci zbytečně často, a o to, abychom zbytečně nezabrali paměť, kterou stejně nepoužijeme
    - Přidáme polovinu již alokovaného
      - A co když už máme alokované např. 1GB na počítači se 4 GB RAM?
      - A co když už jsme na konci přidávání a např. kvůli 128kb bychom tak zabrali 512MB?
  - Častá alokace, která není následována uvolněním naposledy alokovaného bloku, a používání příliš velkých bloků vede k rychlé fragmentaci paměti

- Pro dočasné objekty a kopie proměnných je nejrychlejší alokace na zásobník – ale pozor na životnost a velikost!
  - Protože stačí odečíst rsp namísto volání fce alloc
  - Tj. lokální proměnné a parametry podprogramu

Garbage Collector

- Mj. ponouká programátora k tomu, aby si nedělal hlavu s drahými operacemi (výkon, paměť) jako je new Object
  - A Java si to dokonce vynucuje absencí record/struct
- Způsobuje náhlé, nedeterministické vytížení systému

Kopírování paměti

```
for i:...
   pole[i] = pole[i+1]
```

- Uvedený kód je naivní, protože se kopíruje prvek po prvku
- Daleko efektivnější je použít zkopírování celého bloku paměti, kdy se nebude kopírovat podle velikosti prvku pole, ale podle velikosti slova procesoru
    - Jsou na to standardní funkce,
        - Např. memcpy
        - poskytují je i OS
        - a jsou optimalizované pro danou architekturu – např. pomocí SSE/AVX registrů

Konverze operandů

- Nemíchat operandy různé velikosti
    - Např. konverze mezi single a double

- Integery o velikosti více než 16 bitů se zkonvertují na float rychleji, mají-li znaménko, než když jsou unsigned
- 64bitový double lze typovat na QWORD a paměť porovnat jako dva integery pro konkrétní hodnotu
    - Měl by umět překladač

Využití registrů

- Zásobníková architektura (bytecode) má minimální počet registrů, aby se vědělo, kde zrovna v zásobníku jsme

- x86 není zásobníková architektura, procesor má řadu dalších registrů
  - Lze je využít k předávání parametrů při volání procedur, abychom se vyhnuli zbytečné režii způsobené předáváním parametrů přes zásobník

    - Jazyky pro to mají direktivy jako např. fastcall
    - Např. MS long-mode x64, AMD64 ABI
      - Kromě Rxx registrů používají i registry FPU – XMMx
      - Lze tak pomocí registru předat i 80-bitový Extended (long double)

- Swap
  - Swap jako instrukce bytecodu prohodí dvě slova, která jsou na vrcholu zásobníku
    - Kód narůstá, pokud potřebujeme prohodit slova, které jsou jinde
      - Může být časté, např. viz synchronizace vláken jinak než kritickou sekcí

  - x86 instrukce xchg takové omezení nemá

- Pokud má funkce více parametrů, lze je seskupit do struktury a předat ukazatel na ni
  - Pak se dá předat registrem
  - A neplýtvá se kopírováním hodnot z a do zásobníku
    - Není-li ovšem cílem rychlá alokace struktury

Paralelní vs. sériový kód

```
const size_t size = 100000;
int a[size], b[size], c[size];
...
//Add each pair of elements in arrays
//a and b in parallel and store
//the result in array c.

parallel_for<size_t>(0, size,
                     [&a,&b,&c](size_t i) {
      c[i] = a[i] + b[i];});
             http://msdn.microsoft.com/en-
      us/library/ff601930.aspx#small-loops
```

- Příklad ukazuje kód, kdy jsou náklady na paralelizaci větší než vykonání kódu jedním vláknem
  - Sloučením do jednoho vlákna ostatní vlákna aplikace získají více procesorového času

Java vs. C++

- Daný problém lze v C++ vyřešit efektivněji, než je to možné v Javě
  - By design: Java není self-supportable – bez Assembleru a C++ by neexistovalo JVM ani nativní části runtime knihoven Javy
- Ale na druhou stranu, pro daný problém existuje více programátorů v Javě než v C++